SANDIA REPORT

SAND2014-17752
Unlimited Release
Printed September 2014

Using architecture information and real-time resource state to reduce power consumption and communication costs in parallel applications

James M. Brandt, Karen D. Devine, Ann C. Gentile, Vitus J. Leung, Stephen L. Olivier, Kevin T. Pedretti, and Sivasankaran Rajamanickam

David Bunde (Knox College)

Mehmet Deveci and Ümit V. Çatalyürek (The Ohio State University)

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2014-17752
Unlimited Release
Printed September 2014

Using architecture information and real-time resource state to reduce power consumption and communication costs in parallel applications

James M. Brandt, Karen D. Devine, Ann C. Gentile, Vitus J. Leung,
Stephen L. Olivier, Kevin T. Pedretti, and Sivasankaran Rajamanickam

David Bunde (Knox College)

Mehmet Deveci and Ümit V. Çatalyürek (The Ohio State University)

Abstract

As computer systems grow in both size and complexity, the need for applications and run-time systems to adjust to their dynamic environment also grows. The goal of the RAAMP LDRD was to combine static architecture information and real-time system state with algorithms to conserve power, reduce communication costs, and avoid network contention. We developed new data collection and aggregation tools to extract static hardware information (e.g., node/core hierarchy, network routing) as well as real-time performance data (e.g., CPU utilization, power consumption, memory bandwidth saturation, percentage of used bandwidth, number of network stalls). We created application interfaces that allowed this data to be used easily by algorithms. Finally, we demonstrated the benefit of integrating system and application information for two use cases. The first used real-time power consumption and memory bandwidth saturation data to throttle concurrency to save power without increasing application execution time. The second used static or real-time network traffic information to reduce or avoid network congestion by remapping MPI tasks to allocated processors.

Results from our work are summarized in this report; more details are available in our publications [2, 6, 14, 16, 22, 29, 38, 44, 51, 54].

Acknowledgment

The authors thank Richard Barrett, Erik Boman, Ron Brightwell, Kurt Ferreria, Ryan Grant, Scott Hemmert, James Laros, Steve Plimpton, Andrey Prokopenko, David Robinson, Christian Trott, and Courtenay Vaughan of Sandia National Laboratories for helpful discussions on a range of topics.

We thank our university and industry collaborators as well:

- Bob Alverson, Paul Cassella, Larry Kaplan, Victor Kuhns, Jason Repik, and Jason Schildt (Cray, Inc.);
- Evan Balzuweit, Jonathan Ebbers, Stefan Feer, Austin Finley, Alan Lee, Nickolas Price, Zachary Rhodes, and Matthew Swank (Knox College);
- Michael Showerman, Jeremy Enos, and Joseph Fullop (NCSA);
- Nichamon Naksinehaboon, Narate Taerat, Tom Tucker (Open Grid Computing);
- Torsten Hoefler (UIUC, ETH-Zurich); and
- Sridutt Bhalachandra, Allan Porterfield, and Jan Prins (U. North Carolina).

Power management work as carried out using resources from both the RAAMP LDRD and the Extreme Scale Grand Challenge (XGC) LDRD. including collaboration with Allan Porterfield of RENCI and the University of North Carolina through a subcontract of the XGC project. Multi-node experiments were conducted using the Advanced Architecture Testbeds funded through the ASC program.

The Lightweight Distributed Metric Service (LDMS) software was funded by both the RAAMP LDRD and ASC Facility Operations and User Support (FOUS).

This work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Sandia is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of the RAAMP LDRD (Resource- and Architecture-Aware Mapping and Partitioning) is to collect and deliver static architecture information and real-time system state information, and use it to make decisions about resource usage that can improve application performance and system throughput. This work has increasing importance due to the growing size and complexity of parallel computing systems. Systems can contain hundreds of thousands of nodes, each with dozens of computing cores. Network and memory speeds within systems vary greatly depending on the proximity of the data to the computing core. The size and complexity of such systems increases opportunities for network contention between applications, hardware failures or slowdowns, and scalability challenges for applications. But as we show in this work, system- and application-level tools that exploit static and real-time system information can save power, reduce and avoid network congestion, and, ultimately, reduce application execution time.

Useful static architecture information includes characteristics of the computer hardware: the node/core hierarchy, network topology, and message routes within that topology. It can be used to assign interdependent MPI tasks to "nearby" cores and partition data for greater locality within nodes. Real-time system state information includes CPU and memory utilization, memory bandwidth saturation, network bandwidth usage, and stalls in the network. It can be used to adjust concurrency to save power, rebalance loads with respect to CPU clock speeds and utilization, and map MPI tasks to avoid network congestion.

In this LDRD, we developed capabilities and software tools that resulted in important improvements to system performance. Several examples (with citations for our resulting publications) are listed below.

- We developed strategies to reduce power consumption within nodes by automatically reducing node-level concurrency (number of threads) when memory bandwidth is saturated. We demonstrated this technique both for single-node applications and for parallel multi-node MPI+OpenMP applications. We demonstrated reductions in power consumption of 7.4% in the Lulesh parallel MPI+OpenMP application with no increase in application execution time [51, 29].
- We developed new data collection and aggregation tools in the Lightweight Distributed Metric Service (LDMS), as well as interfaces to deliver the data to algorithms for resource management. The data available includes both static routing information and

network traffic data from Cray's Gemini routers. These new scalable data collection and aggregation strategies provided full system status snapshots within 0.25 seconds on a Cray system with 27648 nodes [2, 16, 54].

- We examined architecture-aware geometric mapping algorithms based on spatial partitioning algorithms. These algorithms reduce application communication costs and execution time by reducing the distance messages travel in the network. Our new architecture-aware geometric mapping in Zoltan2 reduced application execution time by 31% on 64K cores of Cray Cielo for a finite-difference proxy application [22, 6, 38].

- We used our new network performance-counter tools to verify that our mapping algorithms did, indeed, reduce network congestion. We also correlated real-time network data with application performance to identify effective metrics for mapping tasks to cores [44, 22].

- We integrated our real-time performance collection tools with graph-based mapping tools to allow applications to avoid network contention from competing applications. Our resulting dynamic mapping using real-time network data recovered 49% of execution time lost to congestion in a sparse matrix-vector multiplication kernel on a shared computer system [14].

This report contains brief summaries of our research, development and results. For greater details, readers should refer to the publications cited above.

Chapter 2

Dynamic Concurrency Throttling for Power and Energy Reduction

The RAAMP project centers primarily on enabling better decision-making at the application level based on topology-awareness and dynamic information about changing conditions in the operating environment. This chapter presents an example of the impact that such introspection can have in application-level run-time systems for multithreading. In particular, we present the design and implementation of an adaptive run-time system that automatically throttles concurrency using data measured on-line from hardware performance counters. Without source code changes or user intervention, the thread scheduler accurately decides when energy can be conserved by limiting the number of active threads. Initial experiments focused on single-node executions using OpenMP, but we then extended the evaluation to multi-node executions of hybrid MPI+OpenMP codes.

2.1 Problem

The trade-off between performance and energy usage has been a constraint for several generations of commodity microprocessors, and the concern for system-wide power usage is central to the challenge of exascale computing. Decreasing voltage and clock frequency is one common mechanism to reduce power that can result in substantial energy savings for some applications. Intel's Sandybridge chip provides multiple hardware techniques to control frequency, and hardware performance counters to dynamically monitor the chip's energy usage. With these tools, the run-time system can actively participate in the energy/performance trade-off. By limiting the number of active threads, the run-time system can reduce the instantaneous power demand, but the effect on overall energy usage depends on how overall execution time is changed by running on fewer hardware threads. When shared hardware resources (last-level cache, memory or network bandwidth) are oversubscribed, reducing the number of threads will not significantly affect total execution time. In this case, fewer active threads can result in energy savings. However, if there is little interaction between threads and hardware resources are sufficient to support their execution, decreasing the number of threads will result in longer execution time and an increase in the overall energy utilization for the application. Our goal is for the run-time system to determine an appropriate amount of concurrency to save power without negatively impacting execution time.

2.2 RCRTool

Prior to this LDRD, the Resource Centric Reflection Tool (RCRTool) [49] for collection and management of hardware performance counter data was developed at the Renaissance Computing Institute (RENCI) at the University of North Carolina at Chapel Hill. Through this LDRD and the concurrent Extreme Scale Grand Challenge (XGC) LDRD, we worked with RENCI to leverage RCRTool for dynamic run-time system response to online measurements of power usage and performance.

RCRTool consists of two main components: the RCRDaemon, a root-level daemon that continuously samples the hardware performance counters of the CPU, and the RCRBlackboard, an interface that makes the data available at user-level. RCRDaemon samples MSR (Model-Specific Registers) at regular intervals. The sampling frequency, down to a limit of five milliseconds, and the metrics to be measured are selected through a configuration file. The metrics of interest to this project are energy, which was first available on Intel Xeon chips in the SandyBridge generation using the Running Average Power Limit (RAPL) interface [20], and memory concurrency [41], which represents the memory subsystem's capacity for delivering bandwidth across the system under load. The RCRBlackboard publishes data into a shared memory space mapped to the file system, which can be read by applications, run-time systems, and the RCRLogger logging tool.

2.3 Concurrency-throttling Approach

Our approach consists of three steps: 1) monitor system conditions, 2) decide when to adjust concurrency based on observed measurements, and 3) enact required changes. The first step relies on the RCRTool daemon. RCRTool's power-usage and memory-concurrency metrics are required as inputs to the run-time concurrency decision of the run-time system. The run-time system obtains the required data by checking the RCRBlackboard at each scheduling point, or at longer intervals if so configured.

In the second step, the run-time system classifies the assembled data according to system characteristics into one of three classes: a) unsaturated memory concurrency and low power usage, b) saturated memory concurrency and high power usage, or c) moderate concurrency and/or power usage. If the memory concurrency is saturated and power usage is high, the system will decide to decrease the number of active threads and clock down the cores that they occupy. If the memory concurrency is unsaturated and power usage is low, the system will decide to awaken any threads that have previously been deactivated and return their cores to full clock speed. The third, moderate, regime is required to remove any hysteresis effects. Rather than oscillating between turning cores off and on based on small fluctuations in power and memory concurrency, the system will wait until a substantive increase or decrease is observed. The thresholds for high and low power and memory saturation are determined by benchmarking the system to probe for power and memory concurrency limits — benchmarking that need be done only once.

In the last step of the process, the decision made by the run-time system is acted upon. Each thread that is to be deactivated is placed in a local spin loop, and its core's clock modulation register is modified to pass only a fraction of the clock ticks on to the core. Threads are deactivated evenly across the sockets of the system, which alleviates memory pressure evenly. Threads to be reactivated are likewise released from their local spin loops, and their core's clock modulation register set to pass all clock ticks on to the core.

2.4 Implementation in the Qthreads Run-Time System

Qthreads [57] is a cross-platform general-purpose parallel run-time library designed to support lightweight threading and synchronization in a flexible integrated locality framework. Qthreads directly supports programming with lightweight threads and a variety of synchronization methods, including non-blocking atomic operations and potentially blocking full/empty bit (FEB) operations. The Qthreads lightweight threading concept and its implementation are intended to match future hardware environments by providing efficient software support for massive multithreading.

In the Qthreads execution model, lightweight threads (*qthreads*) are created in user-space with a small context and small fixed-size stack. Unlike heavyweight threads such as pthreads, qthreads do not support expensive features like per-thread identifiers, per-thread signal vectors, or preemptive multitasking. Qthreads are scheduled onto a small set of worker pthreads. Logically, a qthread is the smallest schedulable unit of work, such as a set of loop iterations or an OpenMP task, and a program execution generates many more qthreads than it has worker pthreads. Each worker pthread is pinned to a processor core and assigned to a locality domain, termed a *shepherd*. There may be multiple worker pthreads per shepherd, and shepherds may be mapped to different architectural components, e.g., one shepherd per core, one shepherd per shared L3 cache, or one shepherd per processor socket.

OpenMP is supported by Qthreads through the ROSE source-to-source compiler and its XOMP interface [39]. Although Qthreads XOMP/OpenMP support is incomplete, it accepts every OpenMP program accepted by the ROSE compiler. OpenMP directives are outlined and mapped to functions and data structures in the Qthreads library. Explicit tasks and chunks of loop iterations are implemented as qthreads

The MAESTRO extension [50] to Qthreads has implemented alternative thread scheduling mechanisms and polices within the Qthreads run-time system. The Sherwood hierarchical scheduler [42] recognizes that, on non-uniform memory access (NUMA) machines, some threads share a last-level cache and a local memory. Those threads can take advantage of that locality by sharing a LIFO work queue. Constructive cache sharing avoids high-latency accesses and saves memory bandwidth. Work stealing among the queues provides system-wide load balancing.

The Sherwood scheduler has been extended to allow scheduling decisions to be made based on current contention for memory bandwidth. It was integrated with RCRTool to

Configuration	Time	Total Joules	Average Watts
16 Threads - Dynamic	48.4	6860	141.7
16 Threads - Fixed	45.5	7089	155.9
12 Threads - Fixed	48.2	6341	131.5

Table 2.1. Execution time, power, and energy usage of LULESH using dynamic concurrency throttling versus fixed configurations on a single node.

modify scheduling policies to account for dynamic utilization of various shared hardware resources across a multi-socket multi-core node. Based upon the on-line measurements of system resource usage, the run-time system changes the number of worker threads active at any thread initiation point (the beginning of a parallel loop iteration chunk or task instantiation). Internal mechanisms are implemented in Qthreads to allow the number of active threads to vary dynamically to support this ability.

2.5 Evaluation

Initially, we evaluated only single-node usage of our dynamic concurrency throttling [51]. The experiments were carried out on a dual-socket Intel Xeon E5-2680 Sandybridge system (16 cores total) with 64GB of memory. The default clock speed of the processors is 2.70GHz. Intel's Turboboost feature was disabled in the BIOS. The system runs a 3.5.0 pre-release version of the Linux kernel to support additional hardware counter access. Our target program was the OpenMP version of LULESH [36], a mini-application that is a proxy for LLNL's ALE3D hydrodynamics code. As shown in Table 2.1, by dynamically throttling from 16 threads down to 12 threads, the power is reduced by 8%. The execution time is only very slightly degraded compared to the case where 12 threads are used throughout, indicating low overhead for the dynamic throttling mechanism. While running with 12 threads throughout results in higher energy savings, the lack of dynamic control means that users would be responsible for selecting the optimal number of threads. Also, for applications more complex than this simple mini-application, there may be phases of execution where all threads should be used and others where they should not.

Having achieved success at the node level, we then extended our experiments to multi-node executions. In these experiments, each node runs its own run-time system making independent throttling decisions, and the nodes communicate using MPI. The hypothesis was that in a tighly-coupled code like LULESH (in this case, the OpenMP+MPI hybrid version), local run-times would arrive at the same decisions nearly simultaneously. In our experiments, we observed that this was indeed the case. The test system for these experiments was the Compton cluster, in which each node has dual-socket eight-core Intel Xeon E5-2670 processors (16 cores total) running at 2.6 GHz with hyperthreading disabled. The nodes

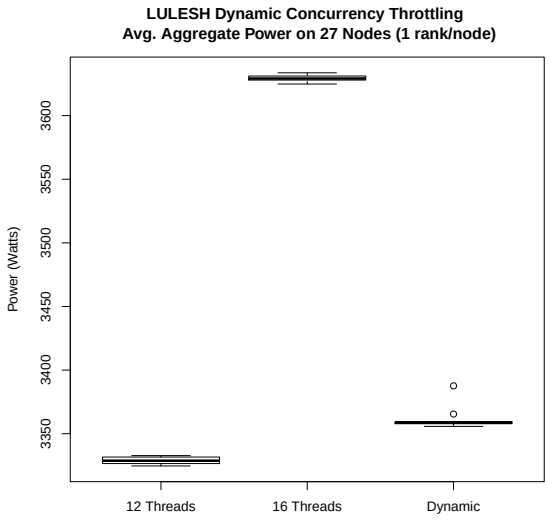


Figure 2.1. Power draw of LULESH using dynamic concurrency throttling versus fixed configurations on 27 nodes.

are connected using Mellanox quad data rate Infiniband. Figures 2.1, 2.2, and 2.3 show the power, energy, and execution time, respectively, of 27-node LULESH executions, using one MPI rank per node. The boxplots are from ten data points per configuration, and show very low variance. Dynamic throttling exhibits 7.4% (270W) power savings versus the 16 thread static configuration, and very near the power usage of the static 12 thread configuration. The excution time with dynamic throttling is actually lower than the 16 thread configuration, and the overall energy usage is also lower. We observe that in the case of multi-node execution, limiting the number of threads not only reduces power, but allows the remaining threads to run faster by relieving the memory pressure.

2.6 Related Work

Over the last decade there has been considerable research into power management, initially for embedded devices and then later for HPC systems and applications. The embedded community has responded to the power challenge through improvements to make the system as well as the applications power-aware [23, 47]. Embedded devices typically have stricter power constraints but less restrictive performance requirements compared to the HPC systems addressed here.

Power management on HPC systems has focused on using the available hardware mechanisms for controlling energy use. The most common mechanism has been dynamic voltage and frequency scaling (DVFS), used in either inter-node [24, 53] or intra-node methods. Our technique is comparable to the intra-node efforts. Early intra-node work by Ge, *et al.* [27] explored opportunities to save energy at fixed frequencies for memory-bound applications.

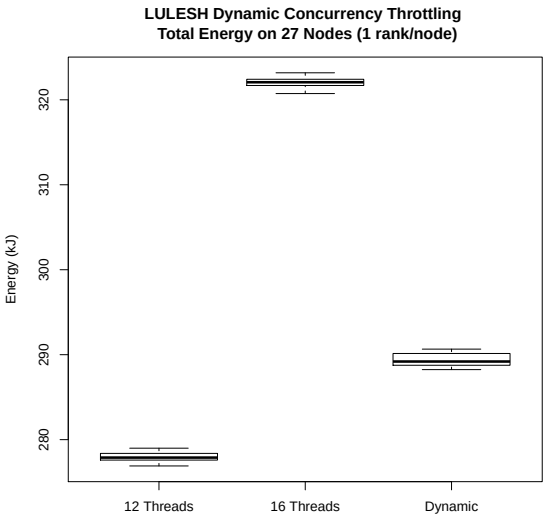


Figure 2.2. Energy usage of LULESH using dynamic concurrency throttling versus fixed configurations on 27 nodes.

Freeh, *et al.* [25] used offline traces to manually divide the work into phases that are run at several frequencies to determine the most energy efficient choice. The Tiwari, *et al.* Green Queue [56] automates the process of finding phases and optimal frequencies using power models. A number of efforts use hardware performance counters [55, 18, 40] to compute optimal off-line settings. Several projects estimate energy usage based on hardware counters with direct correlation including cache access [28], MIPS [34] and CPU stall cycles [35]. While these approaches to estimating and controlling energy use are similar to our work using the new hardware-supplied energy counter, they use DVFS to control power usage. In contrast to clock cycle modulation, which we use to reduce power usage in our work, DVFS has the dual drawbacks of 1) large transition time overheads and 2) global effect on all cores on a chip (currently). Clock cycle modulation can be activated and deactivated very quickly and on a single core basis.

Recently an additional hardware mechanism, power clamping, has been introduced on Intel SandyBridge, along with similar mechanisms on IBM Power 6 and 7 (capping) and AMD Bulldozer (capping and thermal design power limits). Rountree, *et al.* [52] examined the effect of clamping for an HPC application (NAS MG). Their work addressed processor performance variation as HPC moves from performance scheduling to *power scheduling*. Concurrency throttling to match parallelism to available power would operate well within a multi-node power clamping environment. Other work that saves energy by turning off components includes Dynamic Sleep Signal Generator by Youssef [58], which uses off-line traces to predict when functional units can be put to sleep.

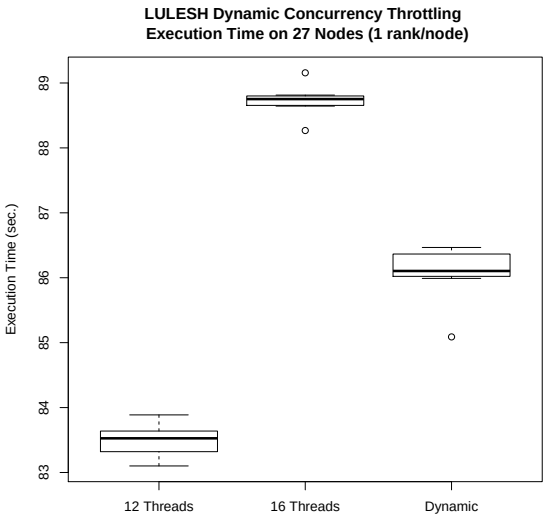


Figure 2.3. Execution time of LULESH using dynamic concurrency throttling versus fixed configurations on 27 nodes.

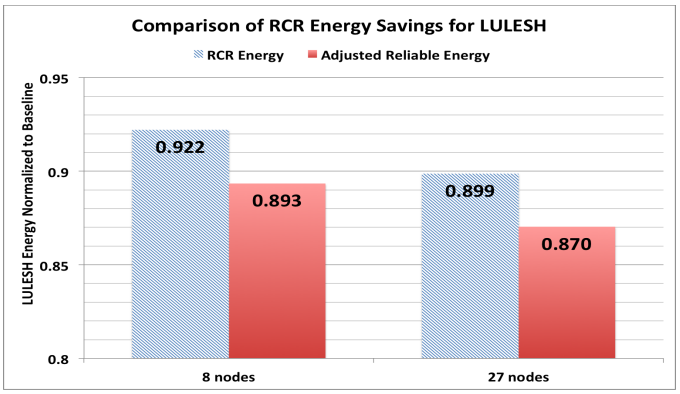


Figure 2.4. An example of the adjusted energy savings from dynamic concurency throttling with LULESH when taking into account failure and recovery costs.

2.7 Conclusions and Impact

The model we developed for our run-time decision-making was fairly simple. With more sophisticated models, there is the potential for even greater power and energy savings and more responsive decision-making. Moreover, there is a reliability benefit to our work. Since our methods can not only save power but also decrease the execution time, there will be fewer checkpoints and, probabilistically, fewer failure and restart events. The power usage of reliability and recovery mechanisms like checkpoint-restart is high, especially since they

use spinning disks. Therefore by decreasing the number of these events, even more energy is saved. Figure 2.4 shows the potential benefits based on production checkpoint and restart parameters [29]. Furthermore, improvements in power controls of future microprocessors will make changing power states easier and more effective. In other words, newer chips can more precisely transition to and operate more efficiently in low power states. These hardware trends will magnify the impact of run-time decisions such as those in this study.

Chapter 3

Data Collection

3.1 Cray Gemini Network Overview

The RAAMP LDRD used the ACES Cielo system at Los Alamos National Laboratory for most of our large-scale testing. Cielo is a Cray XE6 system with 8894 16-core compute nodes (142,304 cores total), providing a peak computational capability of 1.3 PetaFLOPS and aggregate 278 Terabytes of memory capacity [3]. Cielo was an attractive target system for the RAAMP project because of its large scale and its $16 \times 12 \times 24$ 3D torus network topology, which anecdotal evidence suggested was sensitive to task mapping. Targeting Cielo also allowed us to test on other similar systems with a minimal amount effort, such as the Hopper system at NERSC and the Blue Waters system at NCSA. The remainder of this section describes the general architecture of these systems, the network performance counters that are available, and the static routing information that describes how messages are routed across the 3D torus. We utilized the network performance counters and the static routing information to develop and evaluate improved task mapping algorithms.

3.1.1 Cray XE6 architecture

Cray XE systems such as Cielo are based on a custom network processor chip developed by Cray called the Gemini [5]. As its name implies, each Gemini chip connects two separate hosts to the 3D torus. Each host is a one or two socket AMD Opteron node that runs its own Cray Linux operating system instance. Communication between hosts is via explicit message passing, even for hosts connected to the same Gemini. There is no shared memory or cache coherency provided between hosts.

The main functional units of the Gemini network chip are the two network interfaces, each connecting to a different host, and a 48-port router, as shown in Figure 3.1. The 48-ports are organized into seven logical links to implement the 3D torus network topology. One of the logical links is fair-shared between the two hosts connected to the Gemini, allowing each host to send and receive messages on the 3D torus. The other six logical links are used to implement the 3D torus itself, with links in the X+, X-, Y+, Y-, Z+, and Z- directions. The logical links in the X an Z dimensions are each made up of 8 ports each, while the Y dimension logical links are made up of 4 ports (i.e., they are half width). In addition to this

heterogeneity, each port operates at a different speed depending on whether the link is a mezzanine, backplane, or cable link. Further details can be found in [44].

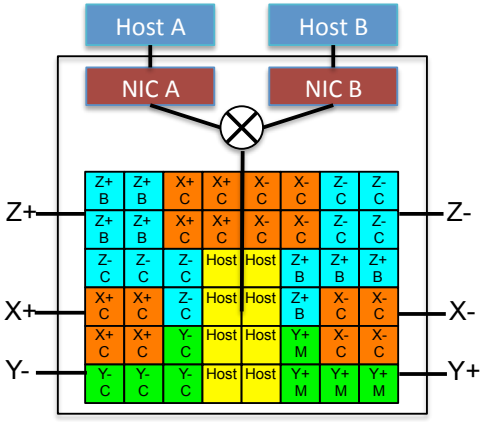


Figure 3.1. Cray XE Gemini building block

In RAAMP, we developed a set of scripts that query the Cray management database and construct a graph of the system with accurate link speeds, accounting for the differences in port direction and type. Vertices in the graph represent either hosts or Gemini chips, and edges represent the network links between them. This full-machine graph can then be used for making task mapping decisions, for example being input into the libtopomap graph-based task mapping library [33]. The graph model of the system, and the knowledge gained by figuring out how to construct it, was leveraged by the LDMS system described in Section 3.2 and in the ResourceOracle route information described in Section 5.2.

The typical usage model of Cray XE systems is that users submit their jobs to a batch scheduling system, which then decides when their jobs are run and which jobs are run simultaneously. All of the jobs running in the system share the Gemini network interface and, in general, a given job is placed on a set of Geminis that are scattered around the overall system and interleaved with Geminis associated with other jobs. This placement often results in network links becoming overloaded (sometimes called hot-spots or network congestion) leading to communication slowdowns. One way to mitigate this effect is to better map the communication pattern of an application to the underlying network topology, thus reducing the load put on the network by the application. This sort of application- and system-aware task mapping is among the key focuses of RAAMP.

3.1.2 Cray Gemini network performance counters

The Gemini provides a set of network interface and router-port network performance counters. We performed a number of empirical experiments to determine what these counters were actually measuring [44], since Cray's documentation was incomplete. We were

particularly interested in the router-port counters, which provide counts for bytes transmitted, packets transmitted, and network stalls for each of the Gemini's 48 router ports.

Unfortunately, when we started the RAAMP project, Cray's software environment had no obvious way to query the router port counters. We leveraged our access to Cray source code to obtain a non-released library, called GPCD (Gemini Performance Counter Daemon), that allowed user-level applications to access the router counters. This driver was part of the open-source, GPL-licensed Cray Gemini Linux driver source code. Once we had the ability to access the counters via the GPCD library, we developed an MPI wrapper library around it that aggregated the router port counters into the seven logical links for each Gemini and scalably gathered application-wide network usage information. We developed post-processing scripts to calculate aggregate statistics such as total bytes injected into the network, total network stalls encountered (an indicator of congestion), and maximum load on any single link.

We used our access to the 3D torus network performance counters for two main purposes in RAAMP.

- We empirically characterized the performance of various task mapping algorithms. For example, we compared the average stalls per byte transmitted for different mapping algorithms; lower values indicate less network congestion.
- We gathered real-time network performance information and used it as input to RAAMP-developed task mapping algorithms. This information enabled a task-mapping algorithm to avoid congested links.

Our library and tools were used for a number of large-scale task mapping experiments, including evaluating our geometric task mapping algorithms (see Chapter 6) on the Cielo platform [22].

3.1.3 Cray Gemini static routing information

In order to compute network path lengths between nodes, we performed experiments to determine the static routing algorithm used by Cray Gemini-based systems. These experiments consisted of sending probe messages from each source node to every possible destination node ($O(n^2)$ messages) and using the per-link network performance counters to trace the path taken for each message [44].

Our experiments showed that routing is performed by traversing the X dimension first, Y next, then Z last. All hops needed in a given dimension are completed before moving onto the next dimension. The shortest number of hops possible is taken in each dimension (i.e., either by traversing in the positive or negative direction around the torus). Also, our experiments showed that the path taken by a request packet from a given source to a destination will, in general, be different from the path of the response packet from the destination back to the

source. This routing makes sense in retrospect, but is not something we would have realized without doing the experiments.

Once we knew the static routing algorithm, we attempted to leverage this information for improving task mapping. We extended the libtopomap [33] input file format to include exact routing information and created a greedy algorithm and a recursive bisection algorithm that incorporated the routing information. The results showed some improvement compared to the baseline algorithms, but there were some performance anomalies that are still being investigated. This work to leverage routing information for task mapping in libtopomap is still ongoing. The exact routing information was also used to develop task mapping algorithms which avoid dynamic network congestion discovered at runtime, as described in Chapter 7 and [14].

3.2 Lightweight Distributed Metric Service (LDMS)

Prior to this LDRD, we developed capabilities to collect high fidelity, system wide, HPC resource utilization information in a lightweight fashion from/about all major system components including compute nodes, file systems, and network. In particular, the Lightweight Distributed Metric Service (LDMS) [54] was initially developed as a proof-of-concept demonstration to determine the feasibility of collecting high fidelity HPC compute-node resource utilization data for use in making run-time partitioning decisions [15]. In this work, we leveraged and extended LDMS' capabilities to make it more suitable for use in large-scale runtime resource allocation decisions. The improvements made to satisfy the needs of the RAAMP project are highlighted below.

3.2.1 LDMS basic architecture

The LDMS framework utilizes a single daemon entity for data sampling, data (or metric) set bundling, aggregation of metric sets from many compute nodes, and storage of data values. The functional differentiation is accomplished by the plugin architecture shown in Figure 3.2. As can be seen in the diagram, sampler plugins allocate memory based on the names, number, and associated value types associated with a particular sampler. Figure 3.3 depicts how a metric set's memory is split into a meta-data and data portion. Meta-data is static and is written in a sampler's memory only upon startup. Every time sampling is performed the contents of the data values are overwritten with those from the new sample; no history is retained. Sample periods are configured during run time on a per-sampler basis. The transport type (socket, RDMA over Infiniband, or RDMA over ugni) is configurable for each LDMS daemon *ldmsd* and must be defined at startup. The framework utilizes a data-pull model in which aggregators pull data on a defined periodic interval from other *ldmsd* daemons (either samplers or aggregators). The pull (or collection) interval is configured on a per-metric-set basis and does not need to match that of the sampler generating the metric set (though in practice, they typically match). The fan-in ratio limit of nodes-to-aggregators

is 16000:1 using RDMA over the ugni transport. An aggregator allocates a storage container for a metric set to be collected; this container is identical to that on the sampling *ldmsd*. As in the case for the sampler, an aggregator reads the meta-data associated with a collected metric set only during the initial read; after the initial read, only the data memory is read from the remote *ldmsd*. Data storage is accomplished by an *ldmsd* using storage plugins. Currently the supported storage types are Comma Separated Value (CSV) flat file, per-metric flat files, and MySQL database. Additionally, to support ease of plotting, we support a derived CSV storage plugin that uses a configuration file to indicate which metrics will be stored either as a rate or value to a CSV file. While a sampler *ldmsd* can also aggregate from other samplers and aggregators, an *ldmsd* that is configured for storage will not store data being sampled by a local sampler plugin.

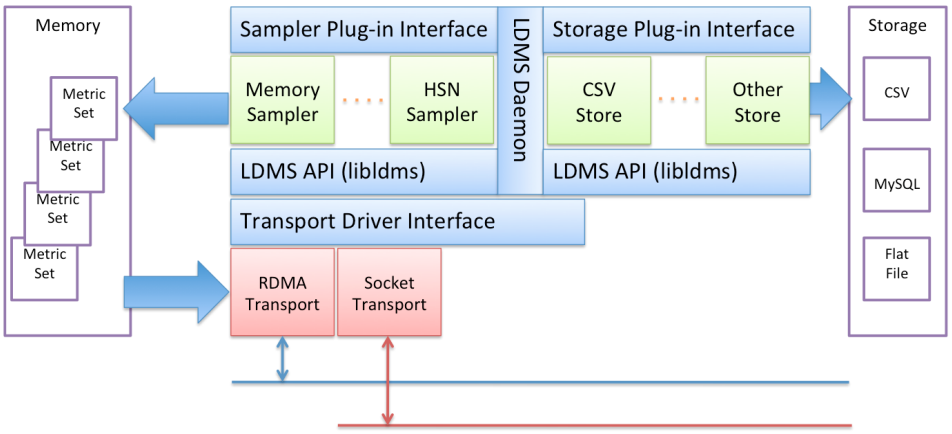


Figure 3.2. LDMS Daemon Plugin Architecture: Green blocks depict plugin modules, both sampler and store, which can be configured during run-time but cannot currently be removed. Red blocks depict transport options (RDMA and Socket) which must be configured at daemon startup. Blue blocks depict the various interfaces and APIs associated with the LDMS Daemon.

3.2.2 LDMS enhancements for large-scale resource decisions

In order to utilize LDMS for large-scale runtime resource mapping decisions, we made a number of enhancements.

- We re-wrote our initial framework, which utilized separate sampler daemons, to use a plug-in architecture. This change decreased the LDMS CPU and memory footprint.
- We incorporated a synchronous mode of operation. In this mode, all nodes' data are sampled at the same time (from the perspective of the nodes), and thus, a global

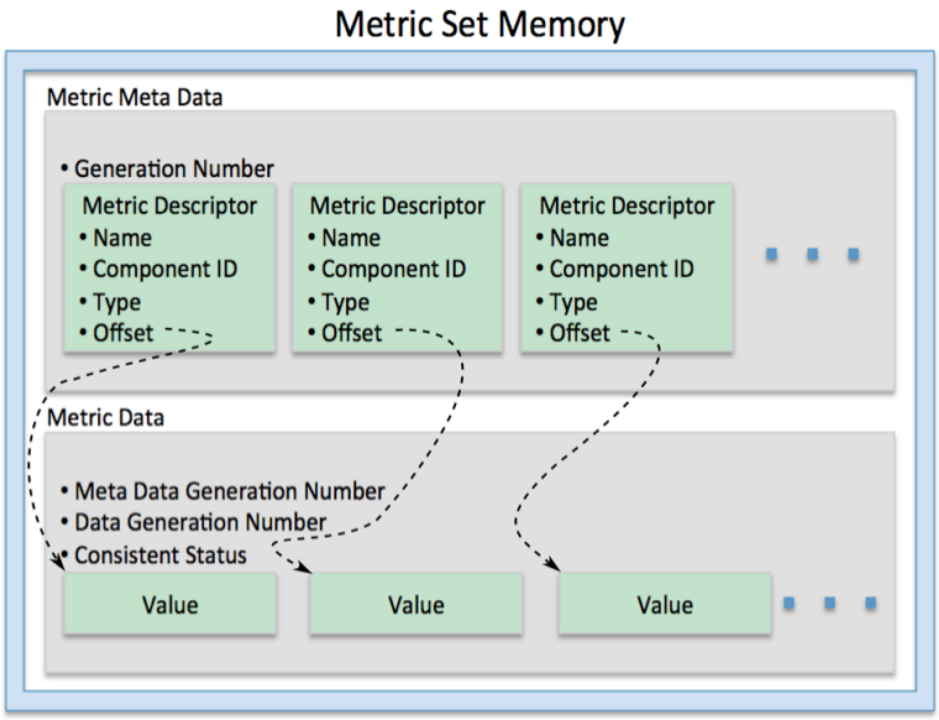


Figure 3.3. LDMS Metric Set Memory Organization: The Metric Meta Data memory chunk contains a block with information for each individual metric. This block contains static information about the metric such as name, component identifier, data type, and the offset that points to the actual data value. The Metric Data memory chunk contains only the data values portion. The Metric Data is the only part of the set to be modified by sampling or to be repeatedly read during collection by an aggregator. The Metric Data is typically around 10% of the total metric set size.

snapshot of system state is obtained. As a result, resource decisions can be made based on a consistent global picture.

- We included the Gemini performance counters in the data set to enable resource decisions based on network contention data.

- We increased fan-in for higher nodes-to-aggregator ratios. This change enabled us to densely aggregate many nodes' data to a few or a single location for the purposes of analyzing and visualizing full system state.

- We created plotting-friendly storage output for visual analyis of congestion data. This capability enabled us to gain understanding of network congestion behavior, including congestion values, durations, and evolution (see Section 3.2.3 below).

3.2.3 Obtaining network traffic data in Gemini architectures with LDMS

As part of a collaboration with NCSA and Cray, LDMS was deployed on NCSA's 27648 node Cray XT/XK system Blue Waters. The goal of the deployment was to provide continuous insight into the high-speed network (HSN) performance counter data in order to gain understanding of network performance issues. This collaboration was in line with the needs of the RAAMP LDRD and provided us unprecedented access to continuous network performance counter data at large scale.

Initially, we read individual GPCD counters and performed link aggregation as described in Section 3.1 and [44]. As part of the the NCSA/Cray/SNL collaboration, Cray developed the GPCDR [1] kernel module, which aggregates Gemini tile counters into a per link value. The GPCDR module also exposes the data to user space via the `/sys` filesystem. We developed a sampler that incorporates this data and corresponding derived rate data, along with system and file-system information, into a single aggregate metric data set.

A sample of the per-node HSN information available from LDMS is shown in condensed form below. This example includes both raw counter data and derived information that can be used to infer network link congestion. The metrics starting with X+ have counterparts in X-, Y+/-, and Z+/- directions which are not shown here.

```
U64 1  nettopo_mesh_coord_X
U64 1  nettopo_mesh_coord_Y
U64 6  nettopo_mesh_coord_Z
U64 511796170434     X+_traffic (B)
U64 11550455465      X+_packets (1)
U64 279915898696     X+_inq_stall (ns)
U64 53317089003      X+_credit_stall (ns)
U64 48               X+_sendlinkstatus (1)
U64 48               X+_recvlinkstatus (1)
U64 13 X+_SAMPLE_GEMINI_LINK_USED_BW (%)
U64 0  X+_SAMPLE_GEMINI_LINK_INQ_STALL (%)
U64 0  X+_SAMPLE_GEMINI_LINK_CREDIT_STALL (%)
```

Mesh coordinates (example: nettopo_mesh_coord_X) are taken in conjunction to define the (X, Y, Z) coordinates of each Gemini router. USED_BW here provides the percentage of total theoretical bandwidth on an incoming link that was used over the last sample interval. INQ_STALL provides the percentage of time, over the last sample interval, that the input queue of the Gemini spent stalled due to lack of credits. CREDIT_STALL provides the percentage of time, over the last sample interval, that traffic could not be sent from the output queue due to lack of credits. These stalls are due to the Gemini network using credit-based flow control. Details of the percentage calculations can be found in [54].

We also developed 2D and 3D visualizations of network contention data in order to help us understand the spatial and temporal characteristics of HSN congestion. An example of Credit Stalls in the X+ direction over all nodes over a 24-hour period is shown in Figure 3.4(a) [2]. From this figure, it can be seen that reasonably high values of stalls can persist from tens of

minutes to many hours. A time snapshot in the Gemini network (3D Torus, $24 \times 24 \times 24$) is shown in Figure 3.4(b) [2].

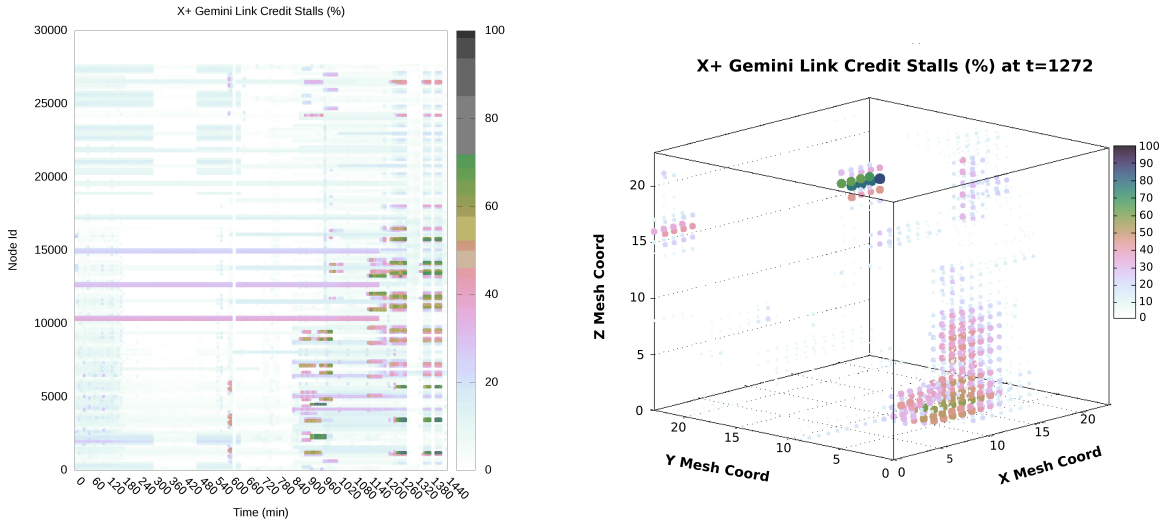


Figure 3.4. Percentage of time spent in Credit Stalls in the X+ direction. (a) Shown for all nodes over a 24 hr period on Blue Waters. Reasonably high values may persist over relatively long time periods. (b) Shown in the Gemini 3D Torus.

In Chapters 5 and 7, we use the LDMS network data to characterize network contention over the routes traversed by an application's traffic and use those characterizations in remapping decisions. The values and durations of the contention measures we see in production on Blue Waters were used as a basis for the values and duration of competing application traffic used in that work. For example, since on Blue Waters we see links with 60% to 70% of their time spent stalled over periods of up to several hours, we recreated a similar congestion scenario in our experiments.

Chapter 4

Representing Dynamic Data in an Architectural Context

Determining and presenting dynamic data in a useful form is non-trivial. The presentation must be architecturally relevant but still not platform-specific. In addition some rebalancing algorithms operate on graph representations, which then require us to extract from the fundamental architecture the relationships and weights that can be used to build a graph and to place the dynamic data on the vertices and edges.

Typical architectural information sources are static and disjoint and do not lend to an integrated approach. For example, network route diagrams do not account for dynamic changes and do not address node-level architecture. Node-level descriptions such as hwloc [43] contain only static values (such as maximum available memory) which do not account for competing processes and do not contain link or neighbor information.

Here we present work-in-progress on a fully generic Data Interface Object which provides a platform-agnostic interface to system data, while still providing data within architecture-aware context. The intent is that the Data Interface Object would be queried for static and/or dynamic data in an architectural context, upon which resource-utilization decisions (e.g., task mapping) could then be made.

4.1 Data Interface Object

The Data Interface Object provides an interface to architecture information and access to data in terms of that architecture. Since the level of detail of the actual physical architecture may not be that desired from the application perspective, both `physical` and `logical` representations of the architecture are defined and built, with mapping between them.

We have preferentially supported hierarchical relationships: `Node`s contain `Socket`s which contain `Core`s etc. This choice was made not only because physical components can typically be considered in such a fashion, but also because the expected use case is that different levels of response would be taken at different levels of the hierarchy. For instance, an initial data mapping might separate data across nodes, while data within a node might be subsequently mapped by taking into consideration processors that share the same memory.

Underlying data structures are `Link`s, typically communication, between `Component`s; and `Attribute`s, which are variable data on `Component`s and `Link`s.

In order to be portable, the data object itself is architecture agnostic. It reads in configuration information specified in XML format and builds the architecture on the fly. Unlike hwloc [43] which has a hardwired set of architectural elements that are allowed, we have defined an XML schema that allows any type to created and assigned.

4.1.1 Platform specifications

At the highest level, the XML schema defines of two types of information. `NamingConvention` is a specification of all the components of a system and their attributes in a manner which allows any given item in the system to be queried via a globally unique UID. The globally unique UID is presented in an SNMP-like dotted string format, which then supports quicker lookup than string comparisons. `Architecture` is a specification of an actual instantiation of a machine in terms of the `NamingConvention`. Excerpts from an XML that illustrate this relationship are below.

```
<naming_convention label="physical" type="dotted_string" catagory="TLCC_p0">
  <component type="node" num="0">
    <attr type="numsocket" num="5" accounting="count" item="socket"/>
      <component type="socket" num="4">
        <attr type="numnumanode" num="1" accounting="count" item="numanode"/>
        <component type="numanode" num="0">
          <attr type="nummemory" num="1" accounting="count" item="memory"/>
          <component type="memory" num="0">
            <attr type="size" num = "0" accounting="static" storage="uint64_t"/>
            <attr type="active" num = "1" accounting="dynamic" storage="ldmsptr"
             metricset="XXX" metric="YYY"/>
          </component>
          <attr type="numcore" num="5" accounting="count" item="core"/>
          <component type="core" num="4">
            <attr type="speed" storage="uint64_t" num = "0" accounting="static"/>
            <attr type="user" storage="ldmsptr" num = "1" accounting="dynamic"
             metricset="procstat" metric="user"/>
            <attr type="sys" storage="ldmsptr" num = "2" accounting="dynamic"
             metricset="procstat" metric="sys"/>
            <attr type="idle" storage="ldmsptr" num = "3" accounting="dynamic"
             metricset="procstat" metric="idle"/>
          </component>
          <attr type="numHT" num="7" accounting="count" item="HT"/>
          <link type="HT" num="6">
            <attr type="start" num = "0" accounting="custom" storage="varchar20"/>
            <attr type="end" num = "1" accounting="custom" storage="varchar20"/>
            <attr type="BW" storage="uint64_t" num = "2" accounting="static"/>
            <attr type="BWused" storage="ldmsptr" num = "3" accounting="dynamic"
             metricset="XXX" metric="YYY"/>
            <attr type="numShared" storage="uint64_t" num = "4" accounting="custom"/>
          </link>
          <attr type="numMemLink" num="9" accounting="count" item="MemLink"/>
          <link type="MemLink" num="8">
            <attr type="start" storage="varchar20" num = "0" accounting="custom" />
            <attr type="end" storage="varchar20" num = "1" accounting="custom"/>
            <attr type="BW" storage="uint64_t" num = "2" accounting="static"/>
            <attr type="BWused" storage="ldmsptr" num = "3" accounting="dynamic"
             metricset="XXX" metric="YYY"/>
            <attr type="numShared" storage="uint64_t" num = "4" accounting="custom"/>
          </link>
        </component>
     </component>
     ...
</naming_convention>
```

```
<architecture label="physical" catagory="TLCC_p0_subset" nc_catagory="TLCC_p0">
  <associations>
    <association label="physical" type="containment">
      <component type="node" num="1" name="N1">
        <component type="socket" num="1" name="S1">
```

```
        <component type="numanode" num="1" name="NN1">
          <component type="core" num="1" name="core1"/>
          <component type="core" num="2" name="core2"/>
          <component type="memory" num="1" name="mem1"/>
        </component>
        <component type="numanode" num="2" name="NN2">
          <component type="core" num="3" name="core3"/>
          <component type="core" num="4" name="core4"/>
          <component type="memory" num="2" name="mem2"/>
        </component>
      </component>
    </component>
  </association>

  <association label="physical" type="HT">
    <link type="HT" num="1" name="HT1" start="NN1" end="NN2" />
    <link type="HT" num="2" name="HT2" start="NN2" end="NN1" />
  </association>
  <association label="physical" type="MemLink">
    <link type="MemLink" num="3" name="MemLink1" start="NN1" end="mem1" />
    <link type="MemLink" num="4" name="MemLink2" start="NN2" end="mem2" />
  </association>
  ...
</architecture>
```

In this case the `NamingConvention` provides sufficient specification to indicate that a legitimate hierarchy is `Nodes` have `Socket`s which have `NumaNode`s which have `Cores`. The dotted string notation consists of colon-separated number pairs with dot separated pairs. The first number in the pair indicates a category in the hierarchy. The second number indicates an instance. Thus the `NamingConvention` indicates that any `Core` will be specified by a dotted string of the form `0.W:5.X:0.Y:4.Z` where `W,X,Y,Z` are variable numbers. The `Architecture` specification is then used to specify the number of `Sockets` and `Cores` per `NumaNode` actually exist and which ones correspond to which instance in the second value of the number pairs. Thus, the two together provide a globally unique identifier for eveything in the system. For example, the third `Core` in this instance will be designated by `0.1:5.1:0.1:4.3`. An `Architecuture` is associated (via field) with a particular `NamingConvention`; multiple `Architectures` could be used with the same `NamingConvention` (e.g., the only change is the number of cores per node).

The dotted string notation allows users to change specificity of the components or the attributes of a component with time without having to invalidate previous naming conventions. New components can be added to the `NamingConvention` with any unused category number; relative catagory numbers are not indicative of anything.

In addition, both `physical` and `logical` `NamingConvention`s are specified. The `physical NamingConvention` is intended to be a physically complete specification of a particular system architecture. The `logical NamingConvention` is a specification of an architecture in terms that a user or application would want to address it. For instance, a user may only be interested in the cores per node and not the intervening layers of the hierarchy. Further, he may want the information presented in terms of communication links between cores, whereas in the actual physical construction, the cores are not the endpoints of the communication links. Typically an application can obtain its rank identity and the core on which it resides and, therefore, accessing communications between processes on cores is more contextually appropriate from the application's perspective.

Excerpts that illustrate the `logical NamingConvention` and `logical Architecture` are below.

```
<naming_convention label="logical" type="dotted_string" catagory="XE6_l0">
  <component type="node" num="0">
    <attr type="numnumanode" num="101" accounting="count" item="numanode"/>
    <component type="numanode" num="100">
      <attr type="nummemory" num="1" accounting="count" item="memory"/>
      <component type="memory" num="0">
        <attr type="size" num = "0" accounting="static" storage="uint64_t"/>
        <attr type="active" num = "1" accounting="dynamic" storage="ldmsptr"
         metricset="XXX" metric="YYY"/>
      </component>
      <attr type="numcore" num="5" accounting="count" item="core"/>
      <component type="core" num="4">
        <attr type="numlink" num="1" accounting="count" item="link"/>
        <link type="link" num="0">
          <attr type="start" num = "0" accounting="custom" storage="varchar20"/>
          <attr type="end" num = "1" accounting="custom" storage="varchar20"/>
          <attr type="BW" storage="uint64_t" num = "2" accounting="static"/>
          <attr type="BWused" storage="ldmsptr" num = "3" accounting="dynamic"
           metricset="XXX" metric="YYY"/>
          <attr type="numShared" storage="uint64_t" num = "4" accounting="custom"/>
        </link>
        <attr type="speed" storage="uint64_t" num = "2" accounting="static"/>
        <attr type="user" storage="ldmsptr" num = "3" accounting="dynamic
         metricset="procstat" metric="user"/>
        <attr type="sys" storage="ldmsptr" num = "4" accounting="dynamic"
         metricset="procstat" metric="user"/>
        <attr type="idle" storage="ldmsptr" num = "5" accounting="dynamic"
         metricset="procstat" metric="user"/>
      </component>
  </component>
 ...
</naming_convention>



<architecture label="logical" catagory="XE6_l0_subset" nc_catagory="XE6_l0">
  <associations>
    <association label="logical" type="containment">
      <component type="node" num="1" name="N1">
        <component type="numanode" num="1" name="NN1">
          <component type="core" num="1" name="core1"/>
          <component type="core" num="2" name="core2"/>
          <component type="memory" num="1" name="mem1"/>
        </component>
        <component type="numanode" num="2" name="NN2">
          <component type="core" num="3" name="core3"/>
          <component type="core" num="4" name="core4"/>
          <component type="memory" num="2" name="mem2"/>
        </component>
        ...
    </association>

    <association label="logical" type="communication">
      <link type="link" num="5" name="comm1" start="core1" end="core2" />
      <link type="link" num="6" name="comm2" start="core1" end="core3" />
      <link type="link" num="7" name="comm3" start="core1" end="core4" />
      <link type="link" num="8" name="comm4" start="core2" end="core1" />
       ...
      <link type="link" num="15" name="comm11" start="core4" end="core2" />
      <link type="link" num="16" name="comm12" start="core4" end="core3" />
    </association>
   ...
</architecture>
```

Finally, `physical`, `logical`, and `logical_to_physical Mapping Architectures` exist. They enable the particular instantiations and mapping between physical and logical components. For example, for the user above, the details of the communication paths are immaterial; thus some communications paths in the `logical Architecture` map to HT links in the `physical Architecture` and some to SHMEM links, as illustrated below.

```
  <architecture label="logical_to_physical" type="mapping" \
        logical_catagory="XE6_l0_subset" physical_catagory="XE6_p0_subset">
  <associations>
    <association label="logical_to_physical" type="mapping">
      <link logical="comm1" physical="MemLink1" />
      <link logical="comm2" physical="HT1" />
      <link logical="comm3" physical="HT1" />
      <link logical="comm4" physical="MemLink1" />
      <link logical="comm5" physical="HT1" />
      <link type="link" logical="comm6" physical="HT1" />
      <link type="link" logical="comm7" physical="HT2" />
      <link type="link" logical="comm8" physical="HT2" />
```

```
      <link type="link" logical="comm9" physical="MemLink2" />
      <link type="link" logical="comm10" physical="HT2" />
      <link type="link" logical="comm11" physical="HT2" />
      <link type="link" logical="comm12" physical="MemLink2" />
      <component logical="N1" physical="N1" />
      <component logical="NN1" physical="NN1" />
      <component logical="NN2" physical="NN2" />
      <component logical="core1" physical="core1" />
      ...
      <component logical="core4" physical="core4" />
      ...
    </association>
  </associations>
</architecture>
```

Not all potential `logical` entities have to be specified. One can select a subset of the full dictionary of components for the system or specify which aspects of the system are of interest (e.g., network only).

4.1.2 Attributes

Attributes are the variable data that will be accessed via the Data Interface Object and upon which resource-utilization decisions will be made. Attributes are defined as either `Static`, `Dynamic`, or `Count`. `Static` values can be specified in the XML file. `Count` variables are automatically filled in when the `Architectures` are parsed and are paired with components (e.g., `numLinks` and `Link`). `Dynamic` is used designate that this variable has a pointer to a dynamic data source. In the example above, the `storage` is defined as `ldmsptr` indicating a pointer to LDMS data (Section 3.2). The `ldmsptr` uses the LDMS interface to get a handle to the LDMS data of the same name on that node (name and metric set in the xml specification). Other possible sources of dynamic data may be used, if the underlying code to those data sources is supplied. The API (Section 4.2) provides functions to obtain these pointers, which can be retained after the whole data structure is released.

A logical component need not have a physical analogue. Where it does, attributes of the physical component are automatically assigned to the logical one (e.g., BW used) if that attibute exists in the logical specification.

4.1.3 NeighborNode information

Neighbor nodes are specified in the XML as a child of the `node`. `NeighborNodes` can therefore have `Attributes` as well. In LDMS, a node can aggregate information from other nodes. This information can then be used as `Attributes` of that `NeighborNode` and enables mapping computations on the node to take into consideration other node's information. `Link` information of a `node` to its `NeighborNodes` is also specified, which enables higher level tools to determine network routes.

4.2 Application and User API

The Data Interface Object is built as a library to be linked into an application with remapping capabilities, or into a standalone application that might calculate static mappings to be used by a scheduler or to construct another intermediate object, such as a graph, for subsequent use.

A Data Interface Object can be built on each node, reading the XML data to build the data structures, and providing the following interface for accessing architecture information and data pointers.

Build/Free the data structures from the XML files:

- `buildDataStructures` — given a set of XML files, create the entire `physical` and `logical hierarchies`.
- `freeDataStructures` — release the entire set of data structures; to be called after the desired data pointers are copied.

Obtain pointers to the internal data structures. These are used to then select the data pointers for the attributes that will be retained:

- `getComponentByOID/Name` — pointers to components can be obtained either by Object Identifier (OID) strings or by Name (e.g., pointer to the `Node` can be obtained by specifying `0.1` or `N1`). Components in either the `physical` or `logical` space can be obtained.
- `getLinkByOID/Name` — analogous to `getComponentByOID/Name`
- `getAttributeByOID/Type` — return a pointer to an `Attribute`, either by its OID or by type (e.g., given a pointer to a `Component` or `Link` and a type defined in the XML, such as `BWUsed`)

Obtain OIDS and DataPtrs. OIDs are used to access other data structures and/or attributes. DataPtrs are used to get a handle into the LDMS data. These functions return memory that must be freed.

- `getLinksFromComponent` — get all the (directional) `Links` from a `Component`.
- `getLinksBetweenComponents` — given two `Components`, get all the `Links` between them.
- `getMatchingComponentsByType` — given an OID with wildcard (e.g., `0.1.1:*`) or type information, return all matching components.

- `getAttributeDataPtr` — get a copy of the data pointer held by an `Attribute`.

Additional requirements for our use cases can be satisfied using these functions. Graphs can be built by querying for the `Links` either among a set of `Component`s (e.g., all `cores`) or from one `Component` to a set of `Component`s of the same or different types (e.g, a node to its neighbors). Calls within the `logical` space allow easy discovery of communication links between `cores` regardless of the actual endpoints in the `physical` architecture. Static or dynamic vertex and edge weights can then be obtained by querying for the data pointers of selected `Attribute`s. For building such inter-node and intra-node graphs, higher functions `buildGraphAA` and `buildGraphAB` are defined which take the vertex, edge, and attribute type information as arguments and call `addVertex` and `addEdge` functions of an external graph interface.

The `logical_to_physical Mapping` information is used to determine the sharing of resources that can lead to contention. From a mapping perspective, this is used when considering resource availability. In the case covered above, several `logical` communication links are actually shared links corresponding to the either the same `physical` analogue `HT` or `MemLink`. This information can be extracted via the `numShared Attribute` of the `Component`s and `Link`s.

4.3 Conclusion

Further development and use of a fully generic data object was put on hold in favor of an object specifically targetting dynamic data values and functions along routes in the Gemini network topology in the Cray XE6 system. This new object and its use are discussed in Chapters 5 and 7, respectively.

Chapter 5

Delivery of Performance Data to Algorithms

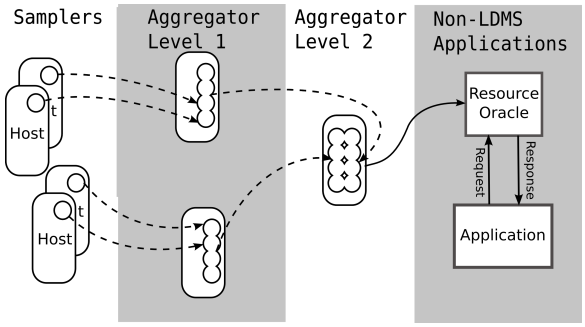
In this chapter, we describe a framework by which data is obtained and aggregated and by which it can be obtained on demand. We specifically target run-time network congestion state to inform resource-allocation decisions, since contention for the shared network fabric can result in substantial performance variation for communication-heavy applications (e.g., Section 3.1). The aggregation of system data is also necessary, because an application's communication often traverses Gemini which are not directly associated with the nodes in its allocation (Section 3.1.1). Aggregated dynamic data is obtained from LDMS (Section 3.2). The *ResourceOracle* is the object with an easy-to-use interface by which this aggregated system data is provided to applications [14].

5.1 Architecture

A high-level diagram of the monitoring and aggregation framework is shown in Figure 5.1. The framework uses LDMS (Section 3.2) to obtain the dynamic state data. Samplers in this figure refer to hosts running monitoring daemons (one per host, including both compute and service nodes). Samplers collect information of interest on the hosts, including network performance counter information. The information is periodically pulled from sampler to aggregator or from aggregator to aggregator (depicted by dashed lines). Aggregators can also be queried by non-LDMS applications for their current data. Figure 5.1 depicts our *ResourceOracle* (`RO`) querying a second-level aggregator.

5.2 Integration of System Data with Route Information

To provide useful, global, network-related information to applications, we integrate collected data with full route information for all pairs of nodes. Applications can then query the `RO` for network information with the full route context. For example, the desired information



(a)

Figure 5.1. High-level diagram of the framework components: LDMS monitoring and aggregation, ResourceOracle, and an application. Rounded rectangles denote LDMS daemons; circles within denote metric sets.

may be the maximum value of USED_BW along the route taken between two nodes.

Route information is built from the individual link information obtained from Cray's "`rtr --phys-routes`" command, while link-type information is obtained using the "`rtr --interconnect`" command. While the first produces a complete listing of pairwise routes including router tile information, the second produces a list of link directions. The second is also the source of the media type (independently used in the USED_BW calculation) which defines the maximum bandwith for each link. Limited output of each is shown below.

```
rtr --phys-routes:
23,24,33,34,43,44,53,54c0-0c0s0g000,01,10,11,25-27,35 ->
06,07,16-22,32c0-0c0s1g000,01,10,11,25-27,35 ->
06,07,16-22,32c0-0c0s2g000,01,10,11,25-27,35 ->
06,07,16-22,32c0-0c0s3g023,24,33,34,43,44,53,54

rtr --interconnect:
c0-0c0s0g0l00[(0,0,0)] Z+ -> c0-0c0s1g0l32[(0,0,1)]   LinkType: backplane
c0-0c0s0g0l01[(0,0,0)] Z+ -> c0-0c0s1g0l21[(0,0,1)]   LinkType: backplane
c0-0c0s0g0l02[(0,0,0)] X+ -> c0-0c1s0g0l02[(1,0,0)]   LinkType: cable11x
c0-0c0s0g0l03[(0,0,0)] X+ -> c0-0c1s0g0l03[(1,0,0)]   LinkType: cable11x
c0-0c0s0g0l04[(0,0,0)] X+ -> c0-0c1s0g0l04[(1,0,0)]   LinkType: cable11x
```

The nid number to cname string (e.g. c0-0c0s0n0) mapping is unique and can be found from `/proc/cray_xt/cname` and `/proc/cray_xt/nid`. For example, `nid00012` is `c0-0c0s6n0` and is associated with Gemini `c0-0c0s6g0`. Given any two nodes, the entire route can be determined from this information. Also, the static path-length metric HOPS (the number links in the route between a pair of nodes) can be computed from this route information.

The *ResourceOracle* is responsible for associating the dynamic monitoring information with the static system information. The `RO` parses the route information once, upon startup. On demand, the `RO` obtains the dynamic information for the relevant links from the current information in the aggregator (Figure 5.1). Bandwidth information is reported by the Gemini performance counters in the incoming direction. Bandwidth and stall values are zero for intra-node communication or for communication between nodes sharing the same Gemini.

5.3 ResourceOracle Interface

The RO provides an interface to applications and, more specifically, to mapping tools by which they can retrieve the information of interest. Mapping tools require information that can be used for graph analysis in the mapping algorithms. The mapping tools identify nodes of interest by their nid numbers as obtained from MPI_Get_processor_name. Needed information includes coordinate information and link weighting information. Thus, the RO provides an API by which Gemini coordinate information and simple functions (e.g., min, max, sum) of well-known metrics (e.g., available bandwidth, credit stalls) of nodes and node pairs may be requested and returned. The queries are simple specifications of the function and metric desired (which are defined in enumerations) and a list of node pairs. The RO listens on a socket for such requests. This interaction is depicted in Figure 5.1, where an application is receiving requested information from the RO.

Chapter 6

Architecture-aware Geometric Task Mapping

6.1 Problem Description

We are developing new architecture-aware strategies for assigning MPI tasks to the cores allocated to applications. In a typical parallel computing environment, jobs are submitted to a batch queuing system. For each job, the batch system selects a set of nodes to be allocated to the job; users have little, if any, control of the allocation they are given. A job's MPI tasks, then, are assigned to the nodes and cores in the allocation with no regard for the application's communication pattern or data locality, nor for the cost of communication between allocated nodes.

Task mapping changes the assignment of MPI tasks to nodes and cores to account for application- and system-specific information. Its goal is to assign interdependent tasks to "nearby" cores in the allocation, so that application communication cost and network congestion are kept low.

Much research has been done on mapping tasks to block-based allocations, such as those on IBM BlueGene systems (e.g., [4, 10, 30, 59]). However, in the Cray systems and clusters commonly used at Sandia, allocations are non-contiguous (i.e., sparse); nodes from any portion of the machine can be assigned to a job without regard to the allocation's shape or locality. Most non-contiguous mapping algorithms represent applications' communication patterns and computers' network topologies as graphs. While finding an optimal graph-based mapping is NP-Complete [33], heuristics can be used to produce good graph-based mappings (e.g., [9, 11, 37, 12, 19, 17]).

We, however, choose to use very inexpensive spatial partitioning algorithms to reorder tasks and processors based on their geometric locality. We use geometric proximity as a proxy for communication costs between processors and for dependence between tasks. Thus, our strategy maps tasks that are "close" to each other geometrically to processors that are "close" to each other in the computing network. Our approach has been designed and tested with torus- and mesh-topology networks. Extensions to other topologies (e.g., the Dragonfly topology in Cray's new Aries network) are the subject of future work.

6.2 Spatial Partitioning Algorithms

Our geometric task mapping methods exploit spatial (coordinate-based) partitioning algorithms commonly used to load-balance parallel computations. These algorithms use only d-dimensional coordinate information to partition data into equally-weighted parts containing data with high geometric locality. Spatial partitioning algorithms are fast and effective for applications that require locality, such as particle methods and contact detection [48].

A well-known and widely used spatial partitioning method is Recursive Coordinate Bisection (RCB) [8]. RCB first computes a cutting plane that is perpendicular to a coordinate axis such that the weight of the data on each side of the cutting plane is equal. The two resulting subdomains are then recursively divided into two equally weighted parts by cutting planes, and so on until the desired number of parts is obtained. The direction of the cutting plane can be chosen by alternating directions, or by cutting perpendicular to the longest dimension in the data. Parallel RCB implementation is available in Zoltan [26].

MultiJagged (MJ) [21] is a spatial partitioning algorithm that can be viewed as a generalization of RCB. Instead of bisecting a dimension into two parts with equal weights, MJ multisects the dimension into k parts (see Figure 6.1 for a 2D example with $k = 3$). The multisection limits the level of recursion in the partitioning algorithm, thereby reducing the global synchronization needed to compute the partition relative to parallel RCB. Like parallel RCB, MJ can migrate data to subsets of processors between recursion levels, but unlike RCB, MJ does this migration only if the cost of global synchronization in future recursion levels will exceed a given threshhold. As a result, MJ is a scalable partitioner for billions of coordinates [21]. An MPI+OpenMP implementation of MJ is available in Zoltan2 [13].

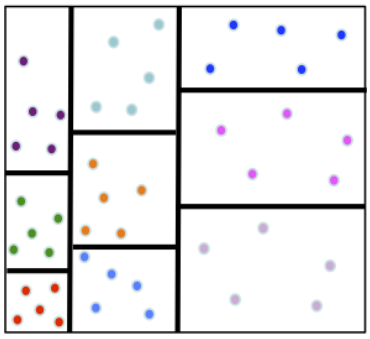


Figure 6.1. Example: a 3×3 MultiJagged (MJ) partition

6.3 Using Spatial Partitioning for Task Mapping

For our task mapping, the goal is to map p MPI tasks to p processors such that interdependent tasks are assigned to "nearby" processors, maintaining geometric locality of

both tasks and processors. First, we obtain coordinates for each processor using the Cray RCA tool. This tool provides (X, Y, Z) coordinates of Gemini routers in the Cray torus network. Two nodes share each router; thus, all cores in those two nodes share the same coordinates. We also obtain (x, y, z) coordinates for each MPI task, taking the average of all data coordinates within each task. We then use a spatial partitioning algorithm twice: once to partition the coordinates of the processors into p parts, and a second time to partition the task coordinates into p parts. Then the MPI task assigned to part i in the task partition is assigned to the processor assigned to part i in the processor partition. Figure 6.2 provides an example of an application with nine tasks (left) mapped onto nine nodes (right); MJ's cut lines are shown for both partitions. By comparing this approach to other simple mapping approaches (e.g., space-filling curves and one-dimensional traversals), we showed that mapping via spatial partitioning produced the smallest application execution times [38]; this work motivated our development of mapping via spatial partitioning in Zoltan2.

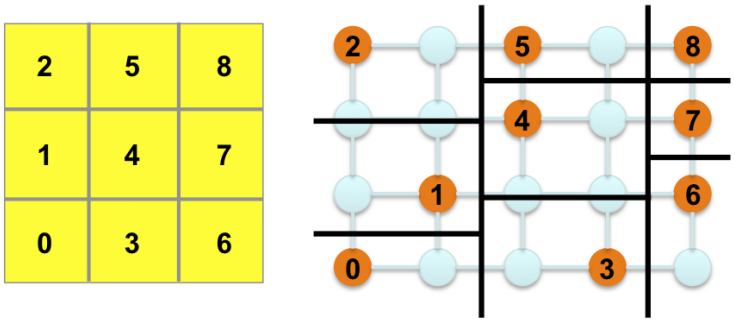


Figure 6.2. Example: tasks (left) are mapped to the allocated (orange) nodes (right) having the same part number in separate 3×3 MultiJagged partitions

The MultiJagged-based task mapping algorithm in Zoltan2 computes mappings using all geometric rotations of the task and processor coordinates. It also shifts the processor coordinates to account for the wrap-around network links in the Cray torus networks. We refer to this algorithm as Geom+R+S (Geometric mapping with Rotations and Shifts).

In experiments comparing Geom+R+S with other mapping methods, we showed that Geom+R+S provided the best application performance in structured applications [22]. Our experiments used the proxy applications MiniGhost [7] and MiniMD [32]. MiniGhost is a finite-difference proxy application that implements a seven-point finite difference stencil and explicit time-stepping scheme across a three-dimensional uniform grid. Its communication pattern reflects the seven-point stencil, with each MPI task communicating with tasks owning data along its subdomain's north, east, south, west, front and back boundaries. MiniMD is a proxy application for molecular dynamics (MD) simulations. In MiniMD's communication pattern, a processor might need to communicate to more than one processor in a direction, based on the cutoff distance used to calculate the force. The experiments were run on the DOE Cielo Cray XE6 at Los Alamos National Laboratory and the Hopper Cray XE6 at NERSC. We used one MPI task per core in all experiments.

For each application, we ran weak scaling experiments to evaluate the effect of mapping on communication and execution time. We compared our geometric method with the applications' default task layout and with the graph-based task mapping library LibTopoMap [33]. For MiniGhost, we also compare with an application-specific grouping of tasks into $2 \times 2 \times 4$ blocks for multicore nodes. The mapping methods we used are listed in Table 6.1.

Method	Abbreviation	Description
No mapping	None	Task i performed by core i
Multicore Grouping	Group	16-task blocks; 2x2x4 tasks per block
Geometric with Rotations + Coordinate Shift	Geom+R+S	Geometric with 36 rotations and torus-aware shifting
LibTopoMap	TopoMap	Graph-based mapping [33]

Table 6.1. Mapping methods used in experiments

Figure 6.3 shows the maximum communication time across processors and total execution time for weak scaling experiments with MiniGhost. The results show that Group reduced execution time by providing greater intranode locality of tasks. But by also accounting for inter-node locality, Geom+R+S provided the best performance. Geom+R+S reduced total execution time on 64K cores by 34% compared to the default MiniGhost mapping, and by 24% compared to Group.

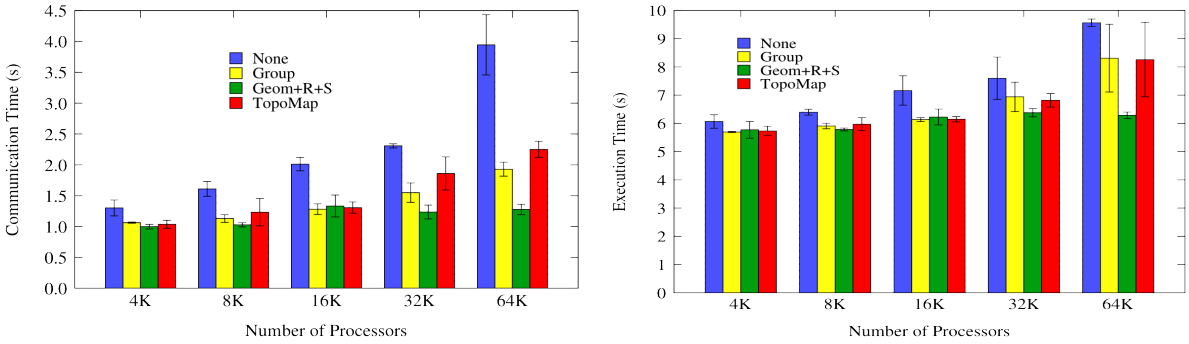


Figure 6.3. Comparisons of maximum communication time (left) and total execution time (right) for several mapping methods in MiniGhost in weak scaling experiments

Similar results were seen for MiniMD (Figure 6.4). Geom+R+S reduced the maximum communication time by 6-27% compared to MiniMD's default mapping on 384 to 6K cores of NERSC's Hopper computer. More information on these experiments is in Deveci et al. [22].

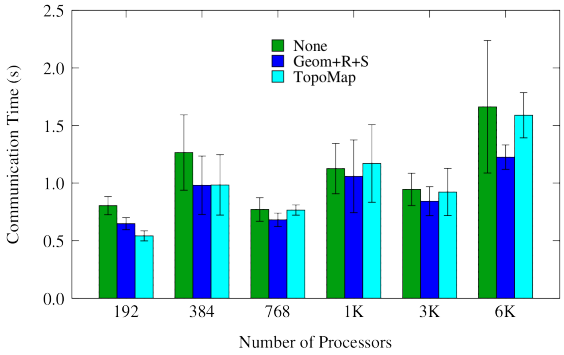


Figure 6.4. Comparison of the maximum communication time for several mapping methods in weak scaling experiments with MiniMD.

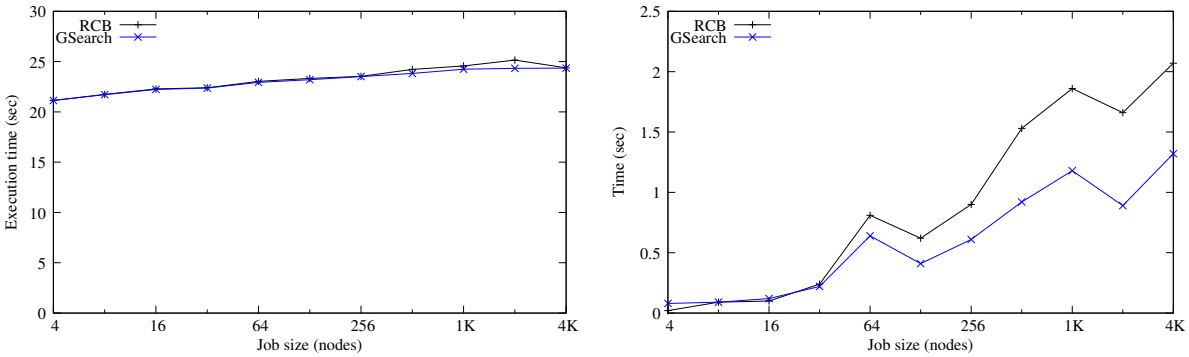
6.4 Local Search

As a refinement to geometric mapping, we examined a variant we call GSearch that includes a post-processing step with a local search to reduce average hops (number of network links between nodes) [6]:

```
do {
  madeSwap = false;
  for 1 ≤ i < num_tasks
    for i < j ≤ num_tasks
      if(swapping tasks i and j reduces average hops) {
        make the swap;
        madeSwap = true;
      }
} while(madeSwap);
```

We evaluated GSearch using experiments with MiniGhost on Cielo. As shown in Figure 6.5(a), using GSearch for task mapping consistently gives lower total execution time than using a prototype, serial version of RCB-based geometric task mapping method, with the improvement increasing with job size up to 2K nodes. In addition, GSearch also reduces the variation in execution time between runs with different allocations. Figure 6.5(b) shows the difference between the longest and shortest runs for each algorithm as a function of job size.

Overall, GSearch provides modest benefits over our prototype RCB implementation. This reinforces our conclusion that geometric methods provide high-quality task mappings. At the same time, the reductions in total execution time and its variance may make GSearch appealing for larger job sizes. In [6], we provide full algorithm descriptions and results, and examine variations and convergence properties.



(a) Average Execution Time

(b) Variation in Execution Time

Figure 6.5. Weak scaling experiments comparing RCB and GSearch using MiniGhost on Cielo: (a) Average total execution time over six runs; (b) Difference between maximum and minimum total execution time.

Chapter 7

Integrating Dynamic Information with Task Mapping

7.1 Approach

In shared computer systems, application performance can be affected negatively by competing applications on the system. Network congestion, in particular, can slow applications as they wait to communicate with other processors in their jobs. Our *architecture-aware* geometric task mapping accounted for the static network topology to reduce congestion, but it did not account for the real-time state of the network. *Resource-aware* mapping that attempts to *avoid* existing network congestion can further improve application performance.

Our resource-aware task mapping relies on several components: real-time data collection and aggregation from LDMS (Section 3.2), application friendly delivery of node-to-node data by the ResourceOracle (Chapter 5), a graph of the MPI tasks' communication pattern, and graph mapping algorithms in the Scotch 6.0 library [45, 46].

Scotch function `SCOTCH_graphMap` takes as input the application's task graph and a graph of the architecture. In the task graph, vertices represent MPI tasks, and weighted edges represent the amount of communication between interdependent tasks. This graph is defined by an application's data and operations. In the architecture graph, vertices represent processors (cores), while weighted edges represent the cost of communication between processors. We construct a complete architecture graph, with an edge between each pair of processors in an allocation. Edge weights are provided by the ResourceOracle. Figure 7.1 includes an examples showing allocated (orange) nodes (left), and the edges incident to one of the allocated nodes in the architecture graph.

Equipped with exact routing information in the Cray's torus network, the ResourceOracle can return aggregated information between pairs of nodes. Aggregation can consist of summing values along a path between nodes, or taking the maximum or minimum value of metrics in links along the path. We experimented with various edge weights in the architecture graph: HOPS (the total number of network links between processors), USED_BW (the maximum percentage of bandwidth used over all links between the processors), and STALLS (the maximum number of credit stall over all links between the processors). The HOPS metric is static, while USED_BW and STALLS are real-time metrics.

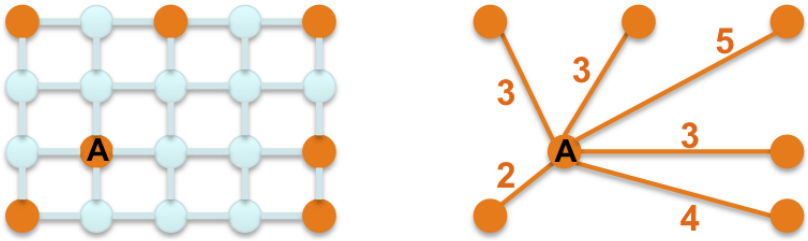


Figure 7.1. Allocated nodes (orange) in the mesh network (left), and the graph edges (right) incident to node A in the architecture graph. Edge weights here represent HOPS, the number of links between allocated nodes. Similar edges exist between all pairs of allocated nodes.

Scotch then looks for a mapping that attempts to minimize the cost of communication in the system, accounting for the amount of data to be sent between tasks as well as the cost of sending that data along the given paths. The output from `SCOTCH_graphMap` is a vector mapping the tasks to the processors.

7.2 Experimental Results

We tested our resource-aware mapping on Sandia's Curie XE6 system, with network dimension $2 \times 2 \times 8$. On Curie, we generated background traffic using a simple bi-directional bandwidth benchmark. Multiple instances of the benchmark were placed in order to target specific links. The maximum percentage of time spent in credit stalls along any link induced by this competing traffic was roughly 68%. The maximum percentage of available bandwidth used along any link was 61%. These generated values are in line with values seen during production conditions on Blue Waters [54, 2].

We then ran an application whose computation was sparse matrix-vector multiplication (SpMV) using the Trilinos [31] solver framekwork. SpMV is a key kernel in many scientific applications. In the congested environment, SpMV with its default task mapping took 19% more time than in an uncongested environment for the same computation. We then applied different task mapping strategies to determine how much of the execution time lost to network congestion they could recover. Specifically, we applied Scotch-based mapping using architecture graphs with different edge weights: HOPS, USED_BW and STALLS. We also applied our static architecture-aware geometric task mapping (MJ) described in Chapter 6.

Experimental results are shown in Figure 7.2. We found that both real-time metrics, USED_BW and STALLS, provided the greatest recovery of execution time, recovering up to 49% of the time lost to congestion. The static mappings (Scotch with the HOPS metric and MJ) provided less benefit; they reduced the distances that SpMV's messages had to travel, but did not avoid contention in the network.

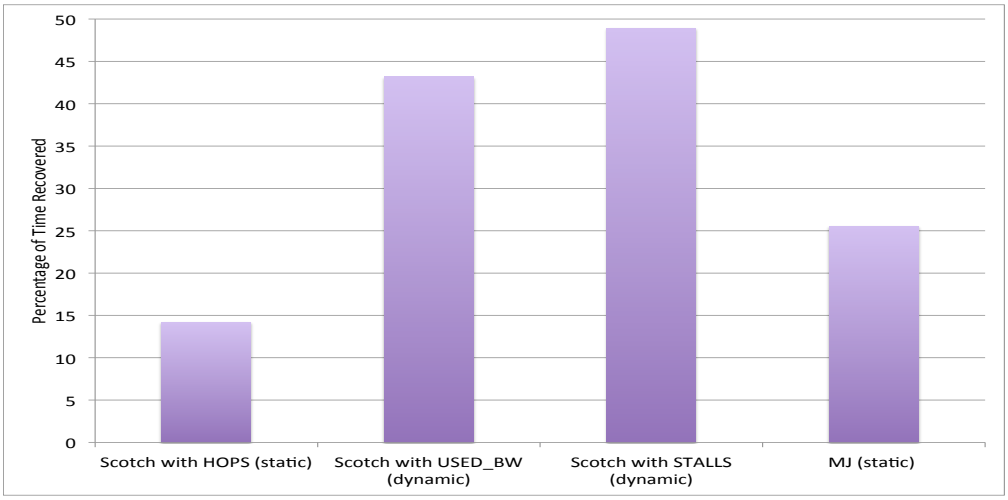


Figure 7.2. Percentage of time due to congestion that is recovered by using various static and dynamic task mapping methods. (Higher is better.)

Chapter 8

Future Work

As new computer systems are designed and move into production use, our tools and algorithms have to be extended and revised for use on those systems. For example, the NNSA/ACES Trinity system will use an Aries interconnect with a DragonFly network topology instead of the Gemini-based torus networks used in this work. The nodes in the DragonFly topology are arranged in groups. Within a group, there is a one-hop link between all nodes in both the x and y dimensions. As a result, messages within a group require at most two hops (one hop in the x direction and one in y). However, groups are connected by a single link, so five hops may be needed for a message between groups to reach its destination. Message routing is not statically defined; instead, messages are routed adaptively, so a message on an adaptive route via another group may require eight hops to reach its destination.

The geometric task mapping algorithm in the present work takes advantage of the torus interconnect and static routing of messages in current-generation Crays. For the non-torus DragonFly topology we may be able to transform the nodes' coordinates so that nodes within a group are closer together in a given dimension than the nodes from another group. This scaling would place interdependent tasks within the same group as much as possible. Representing the adaptive routing in a coordinate-based mapping, though, is still an open problem.

Another interesting challenge would be incorporating real-time network data into our inexpensive geometric mapping. Again, it may be possible to scale distances between nodes by the communication costs returned by the ResourceOracle, so that nodes that have congestion between them appear to be far apart to the geometric mapping algorithm.

Our network performance counters have to be extended to support the Aries interconnect. We have added support for the Aries transport into LDMS (Section 3.2), and have developed a Cray system sampler that collects Aries network metrics. Both of these features have been tested on Cray internal systems, and we will work to deploy them on production systems as they become available.

Scalability of our data collection tools and dynamic strategies needs further investigation as well. Large-scale testing of data collection and aggregation will continue on Cielo and BlueWaters. We are enhancing components of the dynamic task mapping (Chapter 7) framework to scale the capability to larger platforms. We are currently revising the internals

of the ResourceOracle (Chapter 5) to directly access internal data structures of the LDMS aggregator. This improvement will enable the ResourceOracle to obtain dynamic data values currently in the aggregator without incurring additional overhead. For greater scalability in large-scale applications, we will investigate a distributed ResourceOracle.

As of this writing, Cielo has been upgraded to the version of the Cray Linux Environment (CLE) operating system that supports GPCDR and is using a configuration file identical to that used on BlueWaters. We are in discussion with Cielo's administrative staff about the path to installing LDMS and eventually the ResourceOracle on the system as well.

References

[1] Managing System Software for the Cray Linux Environment. Technical Report S-2393-4202, Cray Inc, 2013.

[2] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *Proc. Intl. Conf High Perf. Storage Networking and Analysis (SC14)*, 2014.

[3] Alliance for Computing at Extreme Scale (ACES). Cielo. http://www.lanl.gov/orgs/hpc/cielo/.

[4] G. Almasi, S. Chatterjee, A. Gara, J. Gunnels, M. Gupta, A. Henning, J. Moreira, and B. Walkup. Unlocking the performance of the BlueGene/L supercomputer. In *Proc 2004 ACM/IEEE Conf Supercomputing*, page 57, 2004.

[5] R. Alverson, D. Roweth, and L. Kaplan. The gemini system interconnect. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 83–87, 2010.

[6] E. Balzuweit, D. Bunde, V. Leung, A. Finley, and A. Lee. Local search to improve task mapping. In *Proc 7th Intl Workshop Parallel Prog Models and Systems Software for High-End Computing (P2S2)*. IEEE, 2014.

[7] R. F. Barrett, C. T. Vaughan, and M. A. Heroux. MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Technical Report SAND2012-10431, Sandia National Labs, Albuquerque, NM, 2012.

[8] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans Comput*, C36(5):570–580, 1987.

[9] A. Bhatele, G. Gupta, L. Kale, and I.-H. Chung. Automated mapping of regular communication graphs on mesh interconnects. In *Proc Intl Conf High Performance Computing (HiPC)*, 2010.

[10] A. Bhatele, L. V. Kale, and S. Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proc 23rd Intl Conf Supercomputing*, pages 110–116. ACM, 2009.

[11] S. H. Bokhari. On the mapping problem. *IEEE Trans Comput*, 100(3):207–214, 1981.

[12] S. W. Bollinger and S. F. Midkiff. Heuristic technique for processor and link assignment in multicomputers. *IEEE Trans Comput*, 40(3):325–333, 1991.

[13] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and Ü. Çatalyürek. Zoltan2: Next-generation combinatorial toolkit. Technical Report SAND2012-9373C, Sandia National Labs, 2012.

[14] J. Brandt, K. Devine, A. Gentile, and K. Pedretti. Demonstrating Improved Application Performance Using Dynamic Monitoring and Task Mapping. In *Proc. IEEE Cluster, 1st Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA 2014)*. IEEE, 2014.

[15] J. Brandt, A. Gentile, D. Thompson, and T. Tucker. Develop Feedback System for Intelligent Dynamic Resource Allocation to Improve Application Performance. Technical Report SAND2011-6301, Sandia National Labs, Albuquerque, NM, 2011.

[16] J. Brandt, T. Tucker, A. Gentile, D. Thompson, V. Kuhns, and J. Repik. High Fidelity Data Collection and Transport Service Applied to the Cray XE6/XK6. In *Proc. Cray User's Group (CUG 2013)*, 2013.

[17] T. Chockalingam and S. Arunkumar. Genetic algorithm based heuristics for the mapping problem. *Computers and Operations Research*, 22(1):55–64, 1995.

[18] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proc. 2004 Intl. Symp. Low Power Electronics and Design*, 2004.

[19] I.-H. Chung, C.-R. Lee, J. Zhou, and Y.-C. Chung. Hierarchical mapping for HPC applications. In *Proc Workshop Large-Scale Parallel Processing*, pages 1810–1818, 2011.

[20] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE Intl Symp*, pages 189–194, Aug 2010.

[21] M. Deveci, Ü. V. Çatalyürek, S. Rajamanickam, and K. D. Devine. Multi-Jagged: A scalable multi-section based spatial partitioning algorithm. Technical Report SAND2012-10318C, Sandia National Labs, 2012. Submitted *IEEE Tran. Par. Dist. Sys.*

[22] M. Deveci, S. Rajamanickam, V. Leung, K. Pedretti, S. Olivier, D. Bunde, Ü. V. Çatalyürek, and K. Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *IEEE Int. Parallel & Distributed Processing Symp.* IEEE, 2014.

[23] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP '99: Proc. 17th ACM Symp. Operating Systems Principles*. ACM, 1999.

[24] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. K. Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *Jrnl. Parallel and Distributed Computing*, 68(9), 2008.

[25] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *PPoPP 2005: Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2005.

[26] E. G.Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[27] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *SC05: Proc. 2005 ACM/IEEE Conf High Perf. Networking and Computing*. IEEE Computer Society, 2005.

[28] R. Ge, X. Feng, W. Feng, and K. Cameron. CPU miser: A performance-directed, run-time system for power-aware clusters. In *ICPP 2007: 36th Intl. Conf. Parallel Processing*. IEEE, 2007.

[29] R. Grant, S. Olivier, J. Laros, R. Brightwell, and A. K. Porterfield. Metrics for evaluating energy saving techniques for resilient HPC systems. In *Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW), 2014 IEEE 28th Intl*, pages 1–8, May 2014.

[30] F. Gygi, E. W. Draeger, M. Schulz, B. de Supinski, J. Gunnels, V. Austel, J. Sexton, F. Franchetti, S. Kral, C. Ueberhuber, and J. Lorenz. Large-scale electronic structure calculations of high-Z metals on the BlueGene/L platform. In *Proc 2006 ACM/IEEE Conf Supercomputing*, 2006.

[31] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM TOMS*, 31(3):397–423, 2005.

[32] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Labs, Albuquerque, NM, 2009.

[33] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proc 25th Intl Conf Supercomputing*, pages 75–84. ACM, 2011.

[34] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *SC05: Proc. 2005 ACM/IEEE Conf. High Perf. Networking and Computing*. IEEE Computer Society, 2005.

[35] S. Huang and W. Feng. Energy-efficient cluster computing via accurate workload characterization. In *CCGrid 2009: Proc. 9th IEEE/ACM Intl. Symp. Cluster Computing and the Grid*. IEEE Computer Society, 2009.

[36] Lawrence Livermore National Laboratory. Hydrodynamics challenge problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory, 2010. https://computation.llnl.gov/casc/ShockHydro/LULESH-files/spec.pdf.

[37] S.-Y. Lee and J. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans Comput*, 100(4):433–442, 1987.

[38] V. J. Leung, D. Bunde, J. Ebbers, S. Feer, N. Price, Z. Rhodes, and M. Swank. Task mapping stencil computations for non-contiguous allocations. In *Proc 19th Symp Principals & Practice of Parallel Prog (PPoPP)*. ACM SIGPLAN, 2014.

[39] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, editors, *IWOMP 2010: Proc. 6th Intl. Workshop OpenMP*, volume 6132 of *Lecture Notes in Computer Science*. Springer, 2010.

[40] C. W. Lively, X. Wu, V. E. Taylor, S. Moore, H.-C. Chang, C.-Y. Su, and K. W. Cameron. Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems. *Computer Science - R&D*, 27(4), 2012.

[41] A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE Intl Symp*, pages 66–75, March 2010.

[42] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. OpenMP task scheduling strategies for multicore NUMA systems. *Intl. Jrnl. High Performance Computing Applications*, 26(2), May 2012.

[43] Open MPI Development Team. Portable Hardware Locality (hwloc). `http://www.open-mpi.org/projects/hwloc`.

[44] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and K. S. Hemmert. Using the Cray Gemini performance counters. In *Proc Cray User Group (CUG)*, 2013.

[45] F. Pellegrini. Scotch and LibScotch 6.0 user's guide. Technical report, Universite Bordeaux 1 and LaBRI, 2012.

[46] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[47] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proc. 18th ACM Symp. Operating Systems Principles*, 2001.

[48] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. *Jrnl. Parallel and Distributed Computing*, 50:104–122, 1998.

[49] A. Porterfield, R. Fowler, and M. Y. Lim. RCRTool design document; version 0.1. Technical Report RENCI Technical Report TR-10-01, RENCI, 2010.

[50] A. Porterfield, P. Horst, S. Olivier, R. Fowler, D. O'Brien, K. Wheeler, and B. Viviano. Scheduling OpenMP for Qthreads with MAESTRO. Technical Report TR-11-02, RENCI, 2011. http://www.renci.org/wp-content/uploads/2011/10/TR-11-02.pdf.

[51] A. Porterfield, S. Olivier, S. Bhalachandra, and J. Prins. Power measurement and concurrency throttling for energy reduction in OpenMP programs. In *Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW), 2013 IEEE 27th Intl*, pages 1–8, May 2013.

[52] B. Rountree, D. H. Ahn, B. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *HP-PAC 2012: Proc. 8th Workshop High Performance, Power-Aware Computing*, May 2012.

[53] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. K. Bletsch. Adagio: Making DVS practical for complex HPC applications. In *ICS '09: Proc. 23rd Intl. Conf. Supercomputing*, 2009.

[54] M. Showerman, J. Enos, J. Fullop, P. Cassella, N. Naksinehaboon, N. Taerat, T. Tucker, J. Brandt, A. Gentile, and B. Allan. Large Scale System Monitoring and Analysis on Blue Waters Using OVIS. In *Proc. Cray User's Group (CUG 2014)*, 2014.

[55] D. C. Snowdon, E. L. Sueur, S. M. Petters, and G. Heiser. Koala: a platform for OS-level power management. In *Proc. 2009 EuroSys Conf.*, 2009.

[56] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely. Green queue: Customized large-scale clock frequency scaling. In *CGC '12: Proc. 2nd Intl. Conf. Cloud and Green Computing*, Nov. 2012.

[57] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS '08: Proc. 22nd IEEE Intl. Parallel Dist. Processing Symp.* IEEE, 2008.

[58] A. Youssef, M. Anis, and M. I. Elmasry. Dynamic standby prediction for leakage tolerant microprocessor functional units. In *MICRO 39: Proc. 39th IEEE/ACM Intl. Symp. Microarchitecture*, 2006.

[59] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for Blue Gene/L supercomputer. In *Proc 2006 ACM/IEEE Conf Supercomputing*, 2006.

DISTRIBUTION:

1 Dr. David Bunde
Dept. of Computer Science
Knox College
2 East South Street
Galesburg, Illinois
61401-4999

2 Dr. Ümit V. Çatalyürek
Dept. of Biomedical Informatics
310A Lincoln Tower
1800 Canon Drive
Columbus, OH 43210

1	MS 0801	Tom Klitsner, 9320
1	MS 0801	John Noe, 9328
1	MS 0823	Ann Gentile, 9328
1	MS 0823	James Brandt, 9328
1	MS 1318	Bruce Hendrickson, 1420
1	MS 1318	Robert Hoekstra, 1426
1	MS 1318	Karen Devine, 1426
1	MS 1318	Siva Rajamanickam, 1426
1	MS 1319	Ron Brightwell, 1423
1	MS 1319	Kevin Pedretti, 1423
1	MS 1319	Stephen Olivier, 1423
1	MS 1326	Leann Miller, 1460
1	MS 1327	William Hart, 1464
1	MS 1327	Vitus Leung, 1464
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories